WolfSSL

Report for the Information System Security exam at the Politecnico di Torino

Luca Valentini (278487)

september 2020

Contents

Abstract

This document was intended as a small introductory guide to wolfssl. For the creation of this thesis I used the official wolfssl manual and some github repository related to wolfssl.

1 Communication Security In Embedded Systems

An embedded system can be defined as an autonomous electronic and information system [5]. An embedded system is not really a personal computer (PC), but may resemble an industrial PC. We can see that a standard PC can execute all types of applications as it is designed for general purposes, while an embedded system can only execute a single dedicated application An important property of an embedded system is its ability to communicate with the outside world. The processors used in an embedded system integrate an Ethernet network interface as standard. It is also possible to integrate a wired or wireless internet connection simply by the addition of a small electronic module. Today, controlling an embedded system from a distance has become a reality, and it is completely possible to control an embedded system over the internet via a web browser. This is possible thanks to the power offered by processors designed for embedded systems, and also to the explosion in internet connectivity and the way in which this has become a part of everyday life. IP connectivity basically allows us to control an embedded system remotely over the internet.

Nevertheless, an important problem emerges: communications security in embedded systems in the current situation. With the growing number of embedded applications (aircraft or factory control, transactions, video, etc.), various embedded systems have to communicate with each other over non-secure channels, such as the internet, via wireless connections. There is therefore an enormous risk if data, commands, or sensitive updates are transmitted insecurely over the internet. In order to withstand malicious attacks, the data exchanged must be secured from one end of the transmission to the other. Today many protocols for securing communications (SSH, SSL/TLS, DTLS, IPsec, etc.) are available, and security can be implemented at various levels of the communication stack. However, the greatest obstacles to their use in embedded systems are the limited memory and low processing capacity provided by the platforms of these devices.

Cryptographic algorithms with their many calculations and their significant demands on memory have always presented a barrier to the addition of security to communications among embedded systems. Today, these barriers have become less and less important thanks to the adaptation of security protocols for embedded systems and the use of hardware acceleration techniques, which provide low power processors with the capacity to rapidly execute cryptographic algorithms.

2 WolfSSL

2.1 Introduction

The wolfSSL embedded library is a lightweight TLS library written in ANSI C and targeted for embedded, RTOS, and resource-constrained environments - primarily because of its small size, speed, and feature set; It's an SSL/TLS library optimized to run on embedded platforms. It's free and it has an excellent cross platform support.

WolfSSL supports SSL 3.0, TLS(1.0, 1.1, 1.2, 1.3), and DTLS(1.0, 1.2). It also includes an OpenSSL compatibility interface with the most commonly used OpenSSL functions. WolfSSL is open source, licensed under the GNU General Public License GPLv2.

This library is built for maximum portability and supports the C programming language as a primary interface. It also supports several other host languages, including Java (wolfSSL JNI), C# (wolfSSL C#), Python, and PHP and Perl.

To improve performance it supports hardware cryptography and acceleration on several platforms.

WolfSSL uses the following cryptography libraries:

- wolfCrypt
  - Provides RSA, ECC, DSS, Diffie?Hellman, EDH, NTRU, DES, Triple DES, AES (CBC, CTR, CCM, GCM), Camellia, IDEA, ARC4, HC-128, ChaCha20, MD2, MD4, MD5, SHA-1, SHA-2, SHA-3, BLAKE2, RIPEMD-160, Poly1305, Random Number Generation, Large Integer support, and base 16/64 encoding/decoding.
- NTRU
  - An open source public-key cryptosystem that uses lattice-based cryptography to encrypt and decrypt data.

2.2 Operating system supported

The operating systems supported are:

1. Win32/64
2. Linux
3. Mac OS X
4. Solaris
5. ThreadX
6. VxWorks
7. FreeBSD
8. NetBSD
9. OpenBSD
10. embedded Linux
11. Yocto Linux
12. OpenEmbedded
13. WinCE
14. Haiku
15. OpenWRT
16. iPhone(iOS)
17. Android
18. Nintendo Wii and Gamecube through DevKit-Pro
19. QNX
20. MontaVista
21. NonStop
22. TRON / ITRON / ITRON
23. Micrium C / OS - III
24. FreeRTOS
25. SafeRTOS
26. NXP / Freescale MQX
27. Nucleus
28. TinyOS
29. HP / UX
30. AIX
31. ARC MQX
32. TI - RTOS
33. uTasker
34. embOS
35. INtime
36. Mbed
37. uT - Kernel
38. RIOT
39. CMSIS -RTOS
40. FROSTED
41. Green Hills INTEGRITY
42. Keil RTX
43. TOPPERS
44. PetaLinux
45. Apache Mynewt
46. PikeOS

2.3 WolfSSL's features

- Runtime memory usage between 1-36 kB
- Minimum footprint size of 20-100 kB, depending on build options and operating environment
- OpenSSl compatibility layer
- Hash Functions:
  - MD2
  - MD4
  - MD5
  - SHA-1
  - SHA-224
  - SHA-256
  - SHA-384
  - SHA-512
  - BLAKE2b
  - RIPEMD-160
  - Poly1305
- OCSP, OCSP Stapling, and CRL support
- Block, Stream, and Authenticated Ciphers:
  - AES (CBC, CTR, GCM, CCM, GMAC, CMAC), Camellia, DES, 3DES, IDEA, ARC4, RABBIT, HC-128, ChaCha20
- Public Key Algorithms:
  - RSA, DSS, DH, EDH, ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-RSA, NTRU
- Password-based Key Derivation: HMAC, PBKDF2
- Curve25519 and Ed25519
- ECC and RSA Key Generation
- X.509v3 RSA and ECC Signed Certificate Generation
- PEM and DER certificate support
- Modular cryptography library (wolfCrypt)
- Open Source Project Integrations:
  - MySQL, OpenSSH, Apache httpd, Open vSwitch, stunnel, Lighttpd, GoAhead, Mongoose, and more!
- PKCS#1 (RSA Cryptography Standard) support

- PKCS#3 (Diffie-Hellman Key Agreement Standard) support
- PKCS#5 (Password-Based Encryption Standard) support
- PKCS#7 (Cryptographic Message Syntax - CMS) support
- PKCS#8 (Private-Key Information Syntax Standard) support
- PKCS#9 (Selected Attribute Types) support
- PKCS#10 (Certificate Signing Request - CSR) support
- PKCS#11 (Cryptographic Token Interface) support
- PKCS#12 (Certificate/Personal Information Exchange Syntax Standard) support
- Mutual authentication support (client/server)
- SSL Sniffer (SSL Inspection) Support
- IPv4 and IPv6 support

3 Test case

3.1 Client/Server provided by WolSSL

```
--------------------------------------------------------------
pi@raspberrypi:wolfSSL/wolfssl/examples/server$ ./server -b
SSL VERSION is TLSv1.2
SSL ciphet suite is TLS_ECDHE_RASA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Client message: hello wolfssl!
--------------------------------------------------------------
```

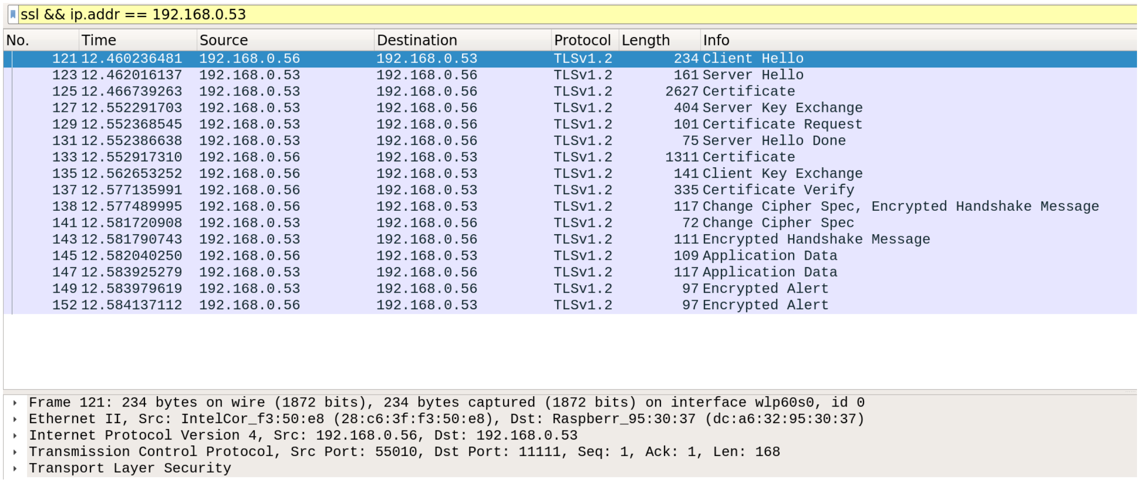
Figure 1: Server TLS

```
----------------------------------------------------------------
luca@luca:wolfSSL/wolfssl/examples/client$ ./client -h
192.168.0.53
SSL VERSION is TLSv1.2
SSL ciphet suite is TLS_ECDHE_RASA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
I hear you fa shizzle!
----------------------------------------------------------------
```

Figure 2: Client TLS

In this example, the server is a simple TLS server that allows only one client connection; after the connection with a client, the server receives an encrypted message from client, it responds and quits. The -b parameter allows the server to bind to any interface instead of localhost only.

After the connection with the server, the client sends a message (hello wolfssl!) and, after the server response, the client quits. The -h parameter allows the client to specify the server address to perform the connection.


ssl && ip.addr == 192.168.0.53

No.	Time	Source	Destination	Protocol	Length	Info
121	12.460236481	192.168.0.56	192.168.0.53	TLSv1.2	234	Client Hello
123	12.462016137	192.168.0.53	192.168.0.56	TLSv1.2	161	Server Hello
125	12.466739263	192.168.0.53	192.168.0.56	TLSv1.2	2627	Certificate
127	12.552291703	192.168.0.53	192.168.0.56	TLSv1.2	404	Server Key Exchange
129	12.552368545	192.168.0.53	192.168.0.56	TLSv1.2	101	Certificate Request
131	12.552386638	192.168.0.53	192.168.0.56	TLSv1.2	75	Server Hello Done
133	12.552917310	192.168.0.56	192.168.0.53	TLSv1.2	1311	Certificate
135	12.562653252	192.168.0.56	192.168.0.53	TLSv1.2	141	Client Key Exchange
137	12.577135991	192.168.0.56	192.168.0.53	TLSv1.2	335	Certificate Verify
138	12.577489995	192.168.0.56	192.168.0.53	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
141	12.581720908	192.168.0.53	192.168.0.56	TLSv1.2	72	Change Cipher Spec
143	12.581790743	192.168.0.53	192.168.0.56	TLSv1.2	111	Encrypted Handshake Message
145	12.582040250	192.168.0.56	192.168.0.53	TLSv1.2	109	Application Data
147	12.583925279	192.168.0.53	192.168.0.56	TLSv1.2	117	Application Data
149	12.583979619	192.168.0.53	192.168.0.56	TLSv1.2	97	Encrypted Alert
152	12.584137112	192.168.0.56	192.168.0.53	TLSv1.2	97	Encrypted Alert

Frame 121: 234 bytes on wire (1872 bits), 234 bytes captured (1872 bits) on interface wlp60s0, id 0
Ethernet II, Src: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8), Dst: Raspberr_95:30:37 (dc:a6:32:95:30:37)
Internet Protocol Version 4, Src: 192.168.0.56, Dst: 192.168.0.53
Transmission Control Protocol, Src Port: 55010, Dst Port: 11111, Seq: 1, Ack: 1, Len: 168
Transport Layer Security

Figure 3: All TLS packets sent

Client IP: 192.168.0.47 Server IP: 192.168.0.53

As you can see from the figure above, the communication is initialized from client with a 'Client Hello'; after TLS handshake, there are 2 messages 'Application Data' sent respectively from the client to the server and from the server to the client, whose content is encrypted. Once the server sends a response to the client, the communication is closed.

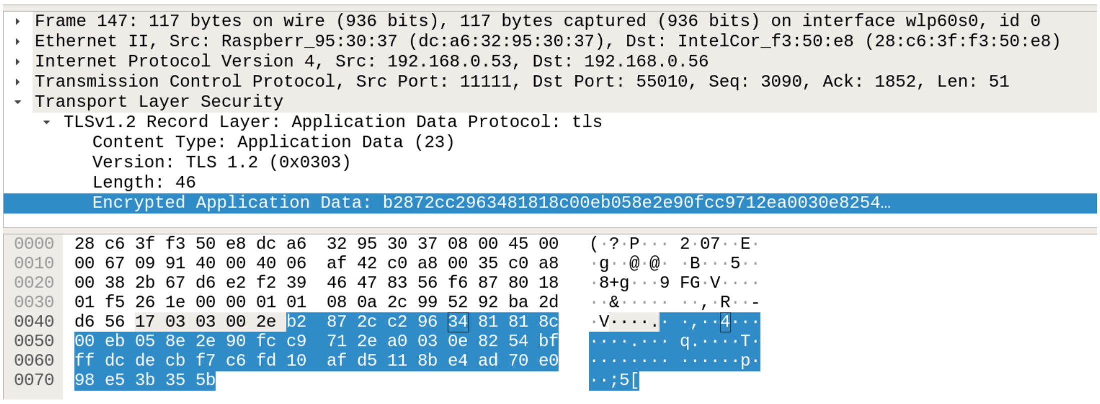


Figure 4: Content of the encrypted message

4 Create a program using WolfSSL

4.1 TCP application

To create a TLS program you can modify your TCP program by adding several TLS functions. To explain the migration from TCP to TLS, I created a simple chat between a client and a server. The code of this program isn't reported in this text but it is available on github.

In the picture there is a wireshark screenshot to see the data traffic of a tcp application; this image with the next sections allows to understand the difference between an application with or without encryption.

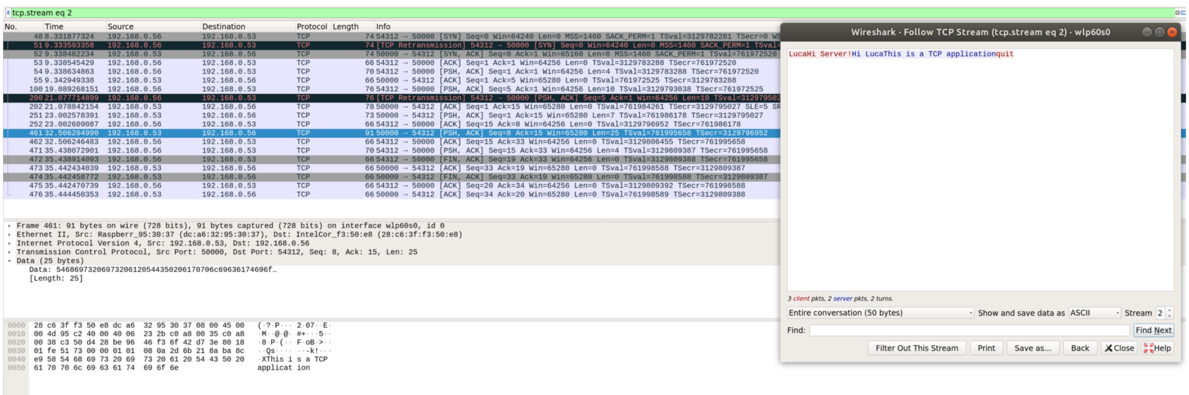


Figure 5: TCP data traffic

4.2 From TCP to TLS

To create a wolfSSL application the first thing that I did was including the wolfSSL API header in my program.

```
#include <wolfssl/ssl.h>
```

After the inclusion of the header files, I initialized the library and the WOLFSSL_CTX calling **wolfSSL_Init**; it is necessary to use the library.

The WOLFSSL_CTX structure contains global values for each SSL connection, including certificate information. To create a new WOLFSSL_CTX there is **wolfSSL_CTX_new()** function. It requires an argument which defines the SSL or TLS protocol for the client or server to use. In my case I used TLS 1.3, so the call is:

```
WOLFSSL_CTX *ctx = wolfSSL_CTX_new(wolfTLSv1_3_server_method());
```

for the server;

```
WOLFSSL_CTX *ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method())
```

for the client.

In the WOLFSSL_CTX the CA (Certificate Authority) can be loaded so that the client is able to verify the server's identity when they start the connection. To load the CA into the WOLFSSL_CTX there is **wolfSSL_CTX_load_verify_locations()**. This function requires three arguments:

- a WOLFSSL_CTX pointer
- a certificate file
- a path value that point to a directory which should contain CA certificates in PEM format.

this function returns SSL_SUCCESS or SSL_FAILURE.

wolfSSL_CTX_load_verify_locations() can be used to verify the certificate of the servers by the client. The server loads a certificate file into the TLS context (WOLFSSL_CTX) through this function:

```
int wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE,
    SSL_FILETYPE_PEM);
```

Then the server must load the private key with:

```
int wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
    SSL_FILETYPE_PEM)
```

After a TCP connection the WOLFSSL object needs to be created and the file descriptor needs to be associated with the session; the instructions are:

```
//Connect to socket file descriptor
WOLFSSL* ssl;
//create WOLFSSL object
ssl = wolfSSL_new(ctx);
wolfSSL_set_fd(ssl,sockfd);
```

After the previous instructions called by client and server, the server waits for a TLS client to initiate the SSL handshake; it waits until a client calls **wolfSSL_connect(ssl)** and then the handshake starts.

Once the connection functions were set, I replaced **read(...)** function with:

```
int wolfSSL_read(WOLFSSL *ssl, void *data, int sz);
```

It read **sz** bytes from the SSL session **ssl** internal read buffer into the buffer **data**. The bytes are removed from the internal receive buffer;

Instead the **write(...)** function is replaced by:

```
int wolfSSL_write(WOLFSSL *ssl, void *data, int sz);
```

It writes **sz** bytes from the buffer, **data**, to the TLS connection, **ssl**.

When the application is over, the WOLFSSL_CTX object and the wolfSSL library must be freed; the instructions are:

```
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

4.3 TLS programs

To develop a TLS application, I started from the TCP chat explained in the previous subsection and I added the wolfSSL function to create an encrypted communication. Next programmes' aim is to show how to create a simple encrypted chat, focusing on the security of applications, rather than on the good practices of the socket and thread applications. Some parts of the code like inclusion and global variables are omitted (wolfSSL object and other variables are stored in global memory);

4.3.1 Iterative program

In the next sections I'll try to explain the code of programs created by me: The iterative wolfSSL program is a very simple chat between two host, a client and a server; it's a ping pong chat where a client can write only if the server write first.

For testing purpose, all the certificates that I use and the server private key were taken from wolfSSL download.
After the initialization of the wolfSSL and the socket, the main function marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept an incoming connection request using accept.

```
int main()
{
    int ret;

    /* Initialize wolfSSL */
    wolfSSL_Init();

    /* Create a socket that uses an internet IPv4 address,
     * Sets the socket to be stream based (TCP),
     * 0 means choose the default protocol. */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){1},
        sizeof(int)) < 0)
    {
        printf("setsockopt(SO_REUSEADDR) failed");
        return -1;
    }

    /* Create and initialize WOLFSSL_CTX */

    if ((ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) ==
        NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL_CTX\n");
```

```
    return -1;
}

/* Load server certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE,
    SSL_FILETYPE_PEM) != SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
            CERT_FILE);
    return -1;
}

/* Load server key into WOLFSSL_CTX */
if (wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
    SSL_FILETYPE_PEM) != SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
            KEY_FILE);
    return -1;
}

/* Initialize the server address struct with zeros */
memset(&servAddr, 0, sizeof(servAddr));

/* Fill in the server address */
servAddr.sin_family = AF_INET;       /* using IPv4    */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
servAddr.sin_addr.s_addr = INADDR_ANY; /* from anywhere */

/* Bind the server socket to our port */
if (bind(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    fprintf(stderr, "ERROR: failed to bind\n");
    return -1;
}

/* Listen for a new connection */
if (listen(sockfd, 1) == -1)
{
    fprintf(stderr, "ERROR: failed to listen\n");
```

```c
    return -1;
}
printf("Waiting for a connection...\n");
/* Accept client connections */
if ((connd = accept(sockfd, (struct sockaddr *)&clientAddr,
    &size)) == -1)
{
    fprintf(stderr, "ERROR: failed to accept the
        connection\n\n");
    return -1;
}

/* Create a WOLFSSL object */
if ((ssl = wolfSSL_new(ctx)) == NULL)
{
    fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
    return -1;
}

/* Attach wolfSSL to the socket */
wolfSSL_set_fd(ssl, connd);

/* Establish TLS connection */
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS)
{
    fprintf(stderr, "wolfSSL_accept error = %d\n",
            wolfSSL_get_error(ssl, ret));
    return -1;
}

printf("Client connected successfully\n");
ClientHandler();
printText("Communication is ended!\n", "System");
ncurses_end();
printf("Shutdown complete\n");

/* Cleanup after this connection */
wolfSSL_free(ssl); /* Free the wolfSSL object        */
close(connd);     /* Close the connection to the client */
/* Cleanup and return */
wolfSSL_CTX_free(ctx); /* Free the wolfSSL context object   */
wolfSSL_Cleanup();   /* Cleanup the wolfSSL environment      */
```

```
    close(sockfd);       /* Close the socket listening for clients
        */
    return 0;            /* Return reporting a success           */
}
```

Listing 1: int main() of iterative TLS server

ClientHandler function reads the username of client using the secure channel and begins the ping pong chat calling writeBuffer() and readBuffer() infinitely until one of the two hosts quit the communication.

```
void ClientHandler()
{
    int ret;
    /*********************** USERNAME */
    /* Read the client username into our buff array */
    XMEMSET(buff, 0, sizeof(buff));
    ret = wolfSSL_read(ssl, buff, sizeof(buff) - 1);
    ncurses_start();
    clearWin();
    if (ret > 0)
    {
        /* Print to stdout any data the client sends */
        strcpy(username, buff);
        char text[256];
        sprintf(text, "Client %s connected successfully", username);
        printText(text, "System");
        printText("***************************\n", "System");
        fflush(stdout);
    }
    else
    {
        printText("ERROR!!", "System");
        close(sockfd);    /* Close the connection to the server */
        pthread_exit(NULL); /* End threaded execution          */
    }
    /**************************** */
    XMEMSET(buff, 0, sizeof(buff));

    while (!is_end && !client_is_end)
    {
        if(writeBuffer() <= 0) break;
        if (!is_end && !client_is_end && readBuffer() <=0) break;
    }
```

```
}
```

Listing 2: void ClientHandler() of iterative TLS server

WriteBuffer() and readBuffer() are omitted because they only call wolfSSL_write(...) and wolfSSL_read(...). The client code too is omitted, because it is very similar to the server code. The goal of this simple program is to check the functioning of wolfSSL library.

```
|--------------------------WolfSSL chat------------------------|
[System] Client Luca connected successfully
[System] ***************************

[Server] Hi Luca!
[Luca] Hi Server!
[Server] Bye Bye
[Luca] Bye
---------------------------------------------------------------|
```

Figure 6: Server TLS

```
|--------------------------WolfSSL chat------------------------|
[Server] Hi Luca!
[Luca] Hi Server!
[Server] Bye Bye
[Luca] Bye
---------------------------------------------------------------|
```

Figure 7: Client TLS

In order to verify the encryption of the communication, I have monitored the packet flow with wireshark.

The IP address of server is **192.168.0.53** and the IP address of client is **192.168.0.49**

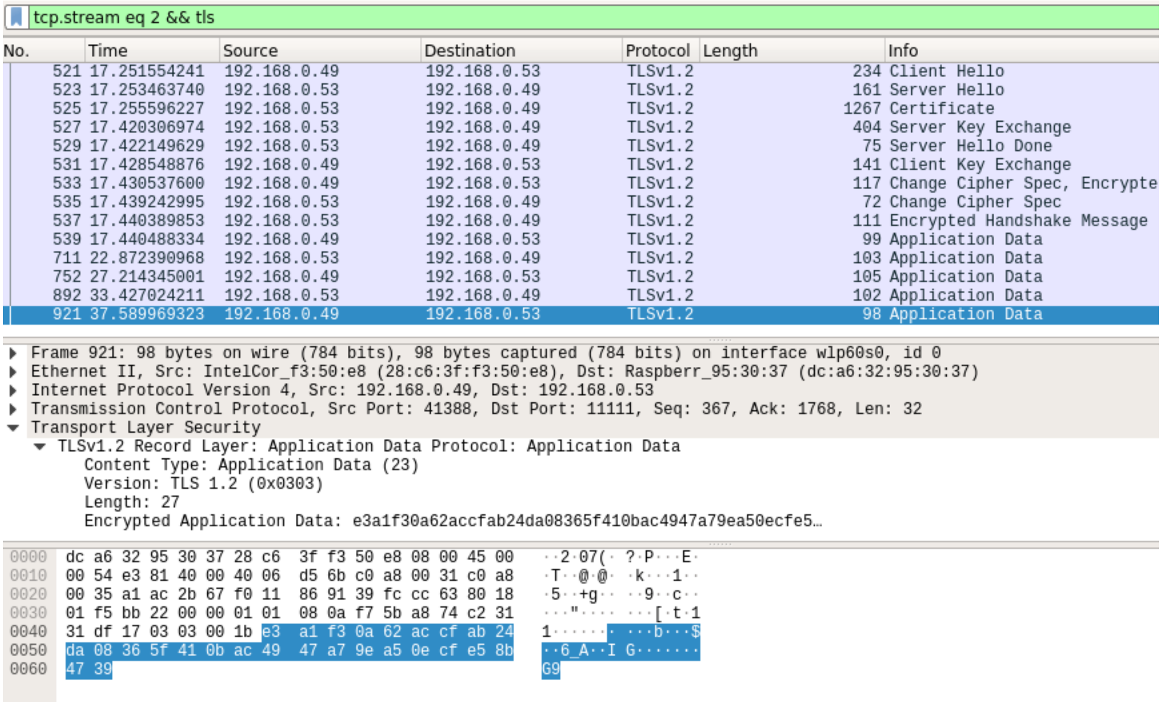


Figure 8: Wireshark's screenshot

4.3.2 Threaded program

The threaded program is a TLS and TCP chat for 10 clients at most. It's a client-server program where multiple clients get connected to the server with a TLS socket and a TCP socket; the two socket per client are used to show the difference, in the same program, between a communication with encryption and a communication without it.

This program is a chat where clients can send messages to other clients; they can choose between a single private message and a public message visible to all clients.

To send a public message a client must only write a message with the keyboard and click enter. This message is sent with a TCP socket to a server; when the server receives a public message through a TCP socket, it sends the message to all TCP socket connected with it, creating a broadcast communication.

When a client wants to send a private message to a connected client, before the message it must put the '#' symbol, followed by the id of the client. To see all the clients id, a client must send 'list' command. The private message is sent with a TLS socket.

4.3.3 Server threaded

The server's main function after the setup of the TLS and TCP socket, it creates a thread to accept the incoming connections from clients.

```
int main()
{
    /* Initialize wolfSSL */
    wolfSSL_Init();

    /* Create a socket that uses an internet IPv4 address,
     * Sets the socket to be stream based (TCP),
     * 0 means choose the default protocol. */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }
    if ((sockfdTCP = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){1},
        sizeof(int)) < 0)
        printf("setsockopt(SO_REUSEADDR) failed");
    if (setsockopt(sockfdTCP, SOL_SOCKET, SO_REUSEADDR, &(int){1},
        sizeof(int)) < 0)
        printf("setsockopt(SO_REUSEADDR) failed");
    /* Create and initialize WOLFSSL_CTX */

    if ((ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) ==
        NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL_CTX\n");
        return -1;
    }

    /* Load server certificates into WOLFSSL_CTX */
    if (wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE,
        SSL_FILETYPE_PEM) != SSL_SUCCESS)
    {
        fprintf(stderr, "ERROR: failed to load %s, please check the
            file.\n",
```

```
            CERT_FILE);
    return -1;
}

/* Load server key into WOLFSSL_CTX */
if (wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
    SSL_FILETYPE_PEM) != SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
            KEY_FILE);
    return -1;
}

/* Initialize the server address struct with zeros */
memset(&servAddr, 0, sizeof(servAddr));

/* Fill in the server address */
servAddr.sin_family = AF_INET;       /* using IPv4    */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
servAddr.sin_addr.s_addr = INADDR_ANY; /* from anywhere */

/* Initialize the server address struct with zeros */
memset(&servAddrTCP, 0, sizeof(servAddrTCP));

/* Fill in the server address */
servAddrTCP.sin_family = AF_INET;          /* using IPv4    */
servAddrTCP.sin_port = htons(DEFAULT_PORT_TCP); /* on
    DEFAULT_PORT */
servAddrTCP.sin_addr.s_addr = INADDR_ANY;  /* from anywhere */

/* Bind the server socket to our port */
if (bind(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    fprintf(stderr, "ERROR: failed to bind\n");
    return -1;
}

/* Bind the server socket to our port */
if (bind(sockfdTCP, (struct sockaddr *)&servAddrTCP,
    sizeof(servAddrTCP)) == -1)
{
```

```
        fprintf(stderr, "ERROR: failed to bind\n");
        return -1;
    }

    /* Listen for a new connection */
    if (listen(sockfd, 10) == -1)
    {
        fprintf(stderr, "ERROR: failed to listen\n");
        return -1;
    }

    /* Listen for a new connection */
    if (listen(sockfdTCP, 10) == -1)
    {
        fprintf(stderr, "ERROR: failed to listen\n");
        return -1;
    }

    if (pthread_create(&Taccept, NULL, acceptConnection, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return -1;
    }

    pthread_join(Taccept, NULL);
    ncurses_end();
    return 0;
}
```

Listing 3: int main() of TLS server

The **void *acceptConnection(void *args)** function is executed by a thread; the aim of this function is to accept incoming connections from TLS and TCP socket. When a TLS and a TCP connection are set, the server assigns an id to the client. After that, two threads are created to read from a TCP socket and a TLS socket.

```
void *acceptConnection(void *args)
{
    int ret;
    ncurses_start();
    clearWin();
    while (1)
```

```c
{
    /* Accept client connections */
    clients[counter].size = sizeof(clients[counter].clientAddr);
    if ((clients[counter].connd = accept(sockfd, (struct
        sockaddr *)&clients[counter].clientAddr,
        &clients[counter].size)) == -1)
    {
        fprintf(stderr, "ERROR: failed to accept the
            connection\n\n");
        return NULL;
    }
    if ((clients[counter].conndTCP = accept(sockfdTCP, (struct
        sockaddr *)&clients[counter].clientAddr,
        &clients[counter].size)) == -1)
    {
        fprintf(stderr, "ERROR: failed to accept the
            connection\n\n");
        return NULL;
    }

    /* Create a WOLFSSL object */
    if ((clients[counter].ssl = wolfSSL_new(ctx)) == NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL
            object\n");
        return NULL;
    }

    /* Attach wolfSSL to the socket */
    wolfSSL_set_fd(clients[counter].ssl,
        clients[counter].connd);

    /* Establish TLS connection */
    ret = wolfSSL_accept(clients[counter].ssl);
    if (ret != SSL_SUCCESS)
    {
        ncurses_end();
        fprintf(stderr, "wolfSSL_accept error = %d\n",
                wolfSSL_get_error(clients[counter].ssl, ret));
        return NULL;
    }
    printText("Client connected successfully\n", "System");
    int *argCounter = malloc(sizeof(*argCounter));
```

```
        if (argCounter == NULL)
        {
            fprintf(stderr, "Couldn't allocate memory for thread
                arg.\n");
            exit(EXIT_FAILURE);
        }

        *argCounter = counter;
        if (pthread_create(&Treader[counter], NULL, readBuffer,
            argCounter))
        {
            fprintf(stderr, "Error creating thread\n");
            fflush(stdout);
            return NULL;
        }
        if (pthread_create(&TreaderTCP[counter], NULL,
            readBufferTCP, argCounter))
        {
            fprintf(stderr, "Error creating thread\n");
            fflush(stdout);
            return NULL;
        }
        counter++;
    }
}
```

Listing 4: void *acceptConnection(void *args) of TLS server

The **void *readBufferTCP(void *args)** function allows to read data from a TCP socket. In the program, there will be as many threads that execute this function as the number of connected clients; it reads messages from socket and it sends their content to all clients. This function will be executed until a client sends a 'quit' message or errors in socket occurred.

```
void *readBufferTCP(void *args)
{
    int ret;
    int id = *((int *)args);
    char output[256] = "";
    while (1)
    {
        memset(clients[id].buffReaderTCP, 0,
            sizeof(clients[id].buffReaderTCP));
        if (read(clients[id].conndTCP, clients[id].buffReaderTCP,
```

```
            sizeof(clients[id].buffReaderTCP)) <= 0)
        {
            fprintf(stderr, "ERROR: failed to read\n");
            pthread_cancel(TreaderTCP[id]);
            return NULL;
        }
        printText(clients[id].buffReaderTCP, clients[id].username);
        if (!strcmp(clients[id].buffReaderTCP, "quit"))
        {
            stopApplication();
            free(args);
            pthread_exit(NULL); /* End threaded execution          */
        }
        memset(output, 0, sizeof(output));
        for (int i = 0; i < counter; i++)
        {
            if (i != id)
            {
                strcat(output, clients[id].username);
                strcat(output, "‘");
                strcat(output, clients[id].buffReaderTCP);
                ret = write(clients[i].conndTCP, output,
                    XSTRLEN(output));
                if (ret <= 0)
                {
                    fprintf(stderr, "ERROR: failed to write\n");
                    pthread_cancel(TreaderTCP[id]);
                    return NULL;
                }
            }
        }
    }
}
```

Listing 5: void *readBufferTCP(void *args) of TLS server

The **void *readBuffer(void *args)** function is similar to the previous function; it reads data from a TLS socket, and it is used for receiving encrypted messages that they will be sent to a single client; this allows to create a single private chat between two clients.

```
void *readBuffer(void *args)
{
    int ret;
```

```c
int id = *((int *)args);
//Read the username
XMEMSET(clients[id].buffReader, 0,
    sizeof(clients[id].buffReader));
ret = wolfSSL_read(clients[id].ssl, clients[id].buffReader,
    sizeof(clients[id].buffReader) - 1);
strcpy(clients[id].username, clients[id].buffReader);

while (1)
{
    /* Read the client data into our buff array */
    XMEMSET(clients[id].buffReader, 0,
        sizeof(clients[id].buffReader));
    if (clients[id].ssl != NULL)
    {
        ret = wolfSSL_read(clients[id].ssl,
            clients[id].buffReader,
            sizeof(clients[id].buffReader) - 1);

        if (ret > 0)
        {
            if (!strcmp(clients[id].buffReader, "list"))
            {
                wolfSSL_write(clients[id].ssl, "Server",
                    XSTRLEN("Server"));
                wolfSSL_write(clients[id].ssl, "Connected
                    clients:", XSTRLEN("Connected clients:"));
                for (int j = 0; j < counter; j++)
                {
                    if (j != id)
                    {
                        char num[50];
                        sprintf(num, "%d", j);
                        wolfSSL_write(clients[id].ssl, num,
                            XSTRLEN(num));
                        wolfSSL_write(clients[id].ssl,
                            clients[j].username,
                            XSTRLEN(clients[j].username));
                    }
                }
            }
            else if (clients[id].buffReader[0] == '#')
            {
```

```
                int dest = clients[id].buffReader[1] - 48;
                    //ASCII
                if (dest <= counter && dest >= 0)
                {
                    char str[255];
                    strcpy(str, "private-");
                    strcat(str, clients[id].username);
                    ret = wolfSSL_write(clients[dest].ssl, str,
                        XSTRLEN(str));
                    strcpy(str, clients[id].buffReader + 2);
                    ret = wolfSSL_write(clients[dest].ssl, str,
                        XSTRLEN(str));
                }
            }
        }
        else
        {
            printText("ERROR READ!!", "System");
            pthread_exit(NULL); /* End threaded execution
                         */
        }
    }
}
free(args);
}
```

Listing 6: void *readBuffer(void *args) of TLS server

4.3.4 Client threaded

The client's main function gets the server's IP address and then it creates a thread to setup the communication.

```
int main(int argc, char **argv)
{
    pthread_t Tclient;
    /* Check for proper calling convention */
    if (argc != 2)
    {
        printf("usage: %s <IPv4 address>\n", argv[0]);
        return -1;
    }
```

```
    ip = argv[1];

    /* create a second thread which executes inc_x(&x) */
    if (pthread_create(&Tclient, NULL, client, NULL))
    {

        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return 1;
    }

    if (pthread_join(Tclient, NULL))
    {

        fprintf(stderr, "Error joining thread\n");
        return 2;
    }
    ncurses_end();
    return 0; /* Return reporting a success          */
}
```

Listing 7: int main(int argc, char **argv) of TLS client

The **void *client(void *args)** function requests to the user an username and then it tries to connect to two different socket; the first one will be used in the public communication, instead the second one will be used to the private communication over a TLS socket.

When the sockets setup is done, the function sends the username to the server over TLS socket. At the end, three other threads will be created to handle the sending and receiving data over TCP and TLS sockets.

```
void *client(void *args)
{
    struct sockaddr_in servAddr;
    struct sockaddr_in servAddrTCP;

    printf("Set your username: ");
    refresh();
    if (!scanf("%s", username))
    {
        fprintf(stderr, "ERROR: failed to get message for
            server\n");
        return NULL;
    }
```

```
ncurses_start();
/* Initialize wolfSSL */
wolfSSL_Init();

/* Create a socket that uses an internet IPv4 address,
 * Sets the socket to be stream based (TCP),
 * 0 means choose the default protocol. */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    fprintf(stderr, "ERROR: failed to create the socket\n");
    return NULL;
}
if ((sockfdTCP = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    fprintf(stderr, "ERROR: failed to create the socket\n");
    return NULL;
}

/* Create and initialize WOLFSSL_CTX */
if ((ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) ==
    NULL)
{
    fprintf(stderr, "ERROR: failed to create WOLFSSL_CTX\n");
    return NULL;
}

/* Load client certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx, CERT_FILE, NULL) !=
    SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
            CERT_FILE);
    return NULL;
}

/* Initialize the server address struct with zeros */
memset(&servAddr, 0, sizeof(servAddr));

/* Fill in the server address */
servAddr.sin_family = AF_INET;       /* using IPv4    */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
```

```
/* Initialize the server address struct with zeros */
memset(&servAddrTCP, 0, sizeof(servAddrTCP));

/* Fill in the server address */
servAddrTCP.sin_family = AF_INET;          /* using IPv4    */
servAddrTCP.sin_port = htons(DEFAULT_PORT_TCP); /* on
    DEFAULT_PORT */

/* Get the server IPv4 address from the command line call */
if (inet_pton(AF_INET, ip, &servAddr.sin_addr) != 1)
{
    fprintf(stderr, "ERROR: invalid address\n");
    return NULL;
}

/* Get the server IPv4 address from the command line call */
if (inet_pton(AF_INET, ip, &servAddrTCP.sin_addr) != 1)
{
    fprintf(stderr, "ERROR: invalid address\n");
    return NULL;
}

/* Connect to the server */

if (connect(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    printText("ERROR: failed to connect", "System");
    return NULL;
}

/* Connect to the server */

if (connect(sockfdTCP, (struct sockaddr *)&servAddrTCP,
    sizeof(servAddrTCP)) == -1)
{
    printText("ERROR: failed to connect", "System");
    return NULL;
}

/* Create a WOLFSSL object */
if ((ssl = wolfSSL_new(ctx)) == NULL)
{
```

```c
        fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
        return NULL;
    }

    /* Attach wolfSSL to the socket */
    wolfSSL_set_fd(ssl, sockfd);
    /* Connect to wolfSSL on the server side */
    if (wolfSSL_connect(ssl) != SSL_SUCCESS)
    {
        fprintf(stderr, "ERROR: failed to connect to wolfSSL\n");
        return NULL;
    }

    strtok(username, "\n");
    len = strnlen(username, sizeof(username));
    /* Send the username to the server */
    if (wolfSSL_write(ssl, username, len) != len)
    {
        fprintf(stderr, "ERROR: failed to write\n");
        return NULL;
    }

    if (pthread_create(&Twriter, NULL, writeBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    if (pthread_create(&Treader, NULL, readBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    if (pthread_create(&TreaderTCP, NULL, readBufferTCP, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }
```

```
    pthread_join(Twriter, NULL);
    pthread_join(Treader, NULL);
    pthread_join(TreaderTCP,NULL);

    /* Cleanup and return */
    wolfSSL_free(ssl);    /* Free the wolfSSL object              */
    wolfSSL_CTX_free(ctx); /* Free the wolfSSL context object     */
    wolfSSL_Cleanup();    /* Cleanup the wolfSSL environment      */
    close(sockfd);        /* Close the connection to the server   */
    printText("Communication is ended!\n Press a button!!!",
        "System");
    getch();
    return NULL;
}
```

Listing 8: void *client(void*args) of TLS client

The **void *writeBuffer(void *args)** function is used to write data in the right socket; it reads data written by the client with read_in() function; if the message written by the client is "list" or the first character is "#", the message will be sent over the TLS socket, otherwise the message will be sent over the TCP socket.

```
void *writeBuffer(void *args)
{
    while (!is_end)
    {
        /* Get a message for the server from stdin */
        memset(Rbuffer, 0, sizeof(Rbuffer));
        read_in();
        len = strnlen(Rbuffer, sizeof(Rbuffer));
        if (XSTRNCMP(Rbuffer, "quit", 4) == 0)
        {
            if (write(sockfdTCP, Rbuffer, len) != len)
            {
                fprintf(stderr, "ERROR: failed to write\n");
                return NULL;
            }
            is_end = 1;
            pthread_cancel(Treader);
            pthread_cancel(TreaderTCP);
        }
        else if (XSTRNCMP(Rbuffer, "list", 4) == 0 || Rbuffer[0] ==
            '#')
```

```
        {
            //wolfSSL_read
            /* Send the message to the server */
            if (wolfSSL_write(ssl, Rbuffer, len) != len)
            {
                fprintf(stderr, "ERROR: failed to write\n");
                return NULL;
            }
        }
        else
        {
            if (write(sockfdTCP, Rbuffer, len) != len)
            {
                fprintf(stderr, "ERROR: failed to write\n");
                return NULL;
            }
        }

        printText(Rbuffer, username);
    }
    return NULL;
}
```

Listing 9: void *writeBuffer(void *args) of TLS client

The **void *readBuffer(void *args)** function reads the username and the message of the sender in the TLS socket, and it prints them in the GUI.

```
void *readBuffer(void *args)
{
    char username[256];
    while (!is_end)
    {
        /* Read the server data into our buff array */
        memset(buffReader, 0, sizeof(buffReader));
        if (wolfSSL_read(ssl, buffReader, sizeof(buffReader) - 1)
            == -1)
        {
            pthread_cancel(Twriter);
            return NULL;
        }
        strcpy(username, buffReader);
        memset(buffReader, 0, sizeof(buffReader));
        if (wolfSSL_read(ssl, buffReader, sizeof(buffReader) - 1)
```

```
            == -1)
        {
            pthread_cancel(Twriter);
            return NULL;
        }
        else
        {
            printText(buffReader, username);
        }
    }
    return NULL;
}
```

Listing 10: void *readBuffer(void *args) of TLS client

The **void *readBufferTCP(void *args)** do the same things of the previous function but it uses a TCP socket.

```
void *readBufferTCP(void *args)
{
    char username[50] = "";
    char output[256] = "";
    int ret;
    while (!is_end)
    {
        memset(buffReaderTCP, 0, sizeof(buffReaderTCP));
        memset(username, 0, sizeof(username));
        memset(output, 0, sizeof(output));
        ret = read(sockfdTCP, buffReaderTCP, sizeof(buffReaderTCP)
            - 1);
        if (ret <= 0)
        {
            pthread_cancel(TreaderTCP);
            return NULL;
        }
        getSenderUsername(buffReaderTCP, username);
        strcpy(output, getSenderData(buffReaderTCP,output));
        printText(output, username);
    }
    return NULL;
}
```

Listing 11: void *readBufferTCP(void *args) of TLS client

4.4 Compile a WolfSSL program

To execute a wolfSSL program you must have installed wolfSSL on your pc, and then you must add **-lwolfssl** on your gcc command.

In this program I used threads and ncurses so I had to add **-pthread** and **-lncurses** flags to gcc command; to optimize the compilation I created a makefile.

4.5 Execute a WolfSSL program (threaded program)

In my project I used a laptop for the client part, and a Raspberry Pi for the server part; they are connected to the same LAN (having only a pc on my home, the clients connected to the server will be on the same machine). In the pictures below you can see the GUI of the client and the server part, and the traffic analyze with wireshark. The IP address of server is **192.168.0.53**; the port for the TLS socket is **11111** and the port for the TCP socket is **11112**. The IP address of clients is **192.168.0.46**
mettere anche i nomi

```
pi@raspberrypi:~/project-wolfssl/code/wolfSSL $./server-tls-threaded
Waiting for a connection...
```

Figure 9: Execute WolfSSL server

```
luca@luca:~/project-wolfssl/code/wolfSSL$./client-tls-threaded
192.168.0.53
Set your username:
```

Figure 10: Execute WolfSSL client

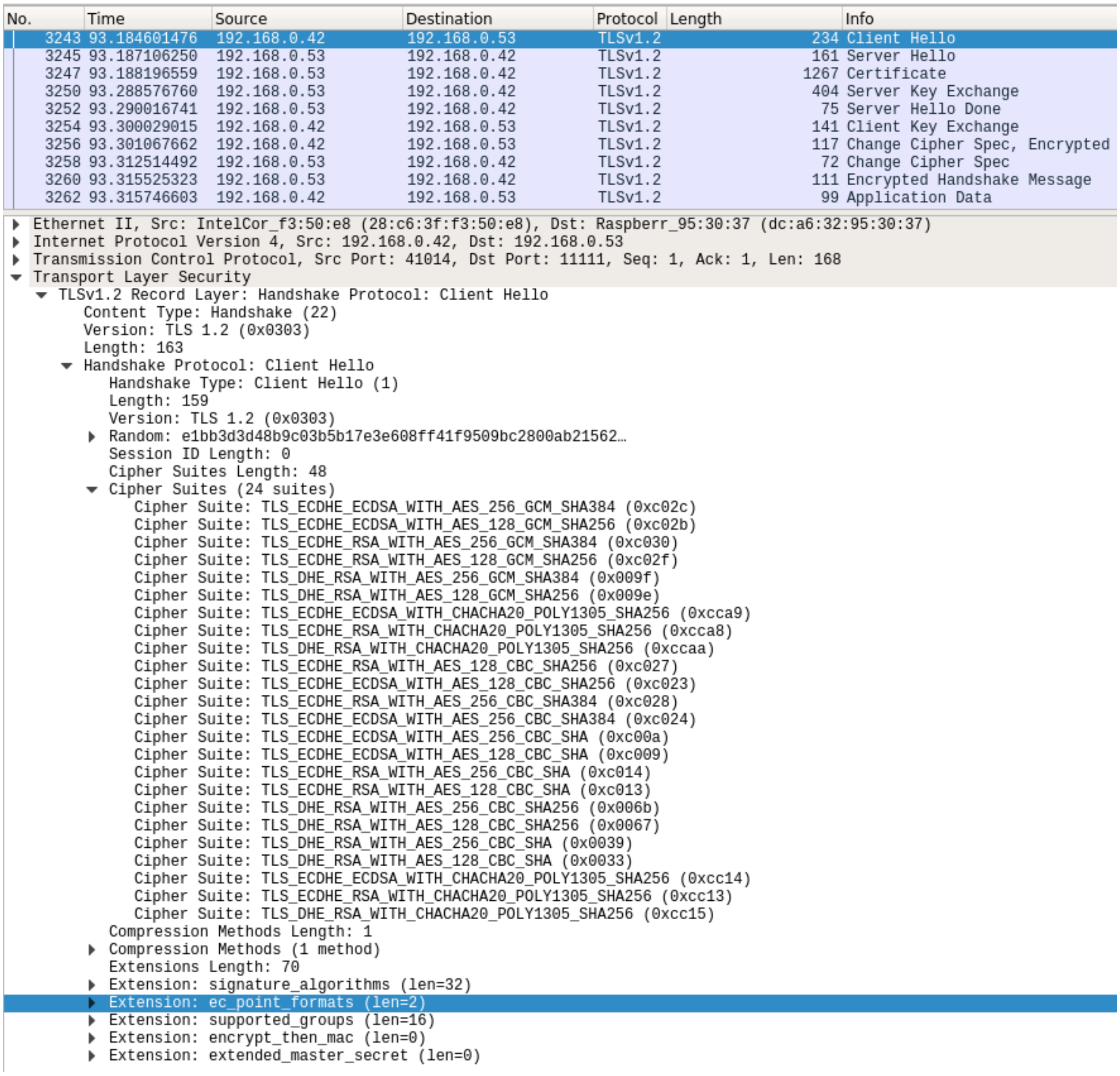
After the execution of the server, a client can connect to it and the ssl handshake starts.

With the hello packet below, the client provides an ordered list of 24 cipher suites that it will support for encryption. The list is in the order preferred by the client, with highest preference first. This list can be modified by the programmer.

In this case, the client provides a list of optional extension which the server can use to take action or enable new features, for example:

- signature_algorithms

– The purpose of this extension is to allow clients to indicate to the server which signature/hash algorithm pairs may be used in digital signatures.



No.	Time	Source	Destination	Protocol	Length	Info
3243	93.184601476	192.168.0.42	192.168.0.53	TLSv1.2	234	Client Hello
3245	93.187106250	192.168.0.53	192.168.0.42	TLSv1.2	161	Server Hello
3247	93.188196559	192.168.0.53	192.168.0.42	TLSv1.2	1267	Certificate
3250	93.288576760	192.168.0.53	192.168.0.42	TLSv1.2	404	Server Key Exchange
3252	93.290016741	192.168.0.53	192.168.0.42	TLSv1.2	75	Server Hello Done
3254	93.300029015	192.168.0.42	192.168.0.53	TLSv1.2	141	Client Key Exchange
3256	93.301067662	192.168.0.42	192.168.0.53	TLSv1.2	117	Change Cipher Spec, Encrypted
3258	93.312514492	192.168.0.53	192.168.0.42	TLSv1.2	72	Change Cipher Spec
3260	93.315525323	192.168.0.53	192.168.0.42	TLSv1.2	111	Encrypted Handshake Message
3262	93.315746603	192.168.0.42	192.168.0.53	TLSv1.2	99	Application Data

```
▸ Ethernet II, Src: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8), Dst: Raspberr_95:30:37 (dc:a6:32:95:30:37)
▸ Internet Protocol Version 4, Src: 192.168.0.42, Dst: 192.168.0.53
▸ Transmission Control Protocol, Src Port: 41014, Dst Port: 11111, Seq: 1, Ack: 1, Len: 168
▾ Transport Layer Security
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 163
    ▾ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 159
        Version: TLS 1.2 (0x0303)
      ▸ Random: e1bb3d3d48b9c03b5b17e3e608ff41f9509bc2800ab21562…
        Session ID Length: 0
        Cipher Suites Length: 48
      ▾ Cipher Suites (24 suites)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
          Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
          Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xccaa)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x006b)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)
          Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)
          Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc15)
        Compression Methods Length: 1
      ▸ Compression Methods (1 method)
        Extensions Length: 70
      ▸ Extension: signature_algorithms (len=32)
      ▸ Extension: ec_point_formats (len=2)
      ▸ Extension: supported_groups (len=16)
      ▸ Extension: encrypt_then_mac (len=0)
      ▸ Extension: extended_master_secret (len=0)
```

Figure 11: Client Hello

In the Server Hello packet, the server has selected cipher suite 0xc030 (TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384) from the list of options given by the client.

Random is 32-byte random number used to generate the Master Secret.

The session identifier is a unique number to identify the session for the corresponding connection with the client.

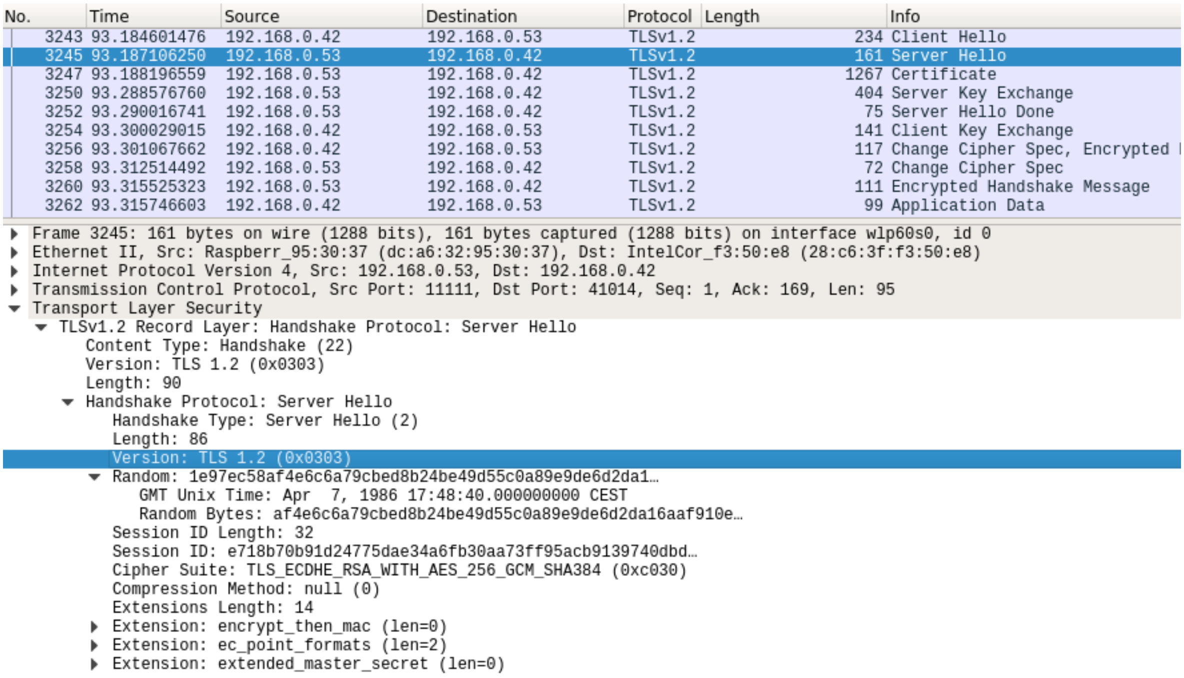


Figure 12: Server Hello

The server sends the client a list of certificates to authenticate itself.

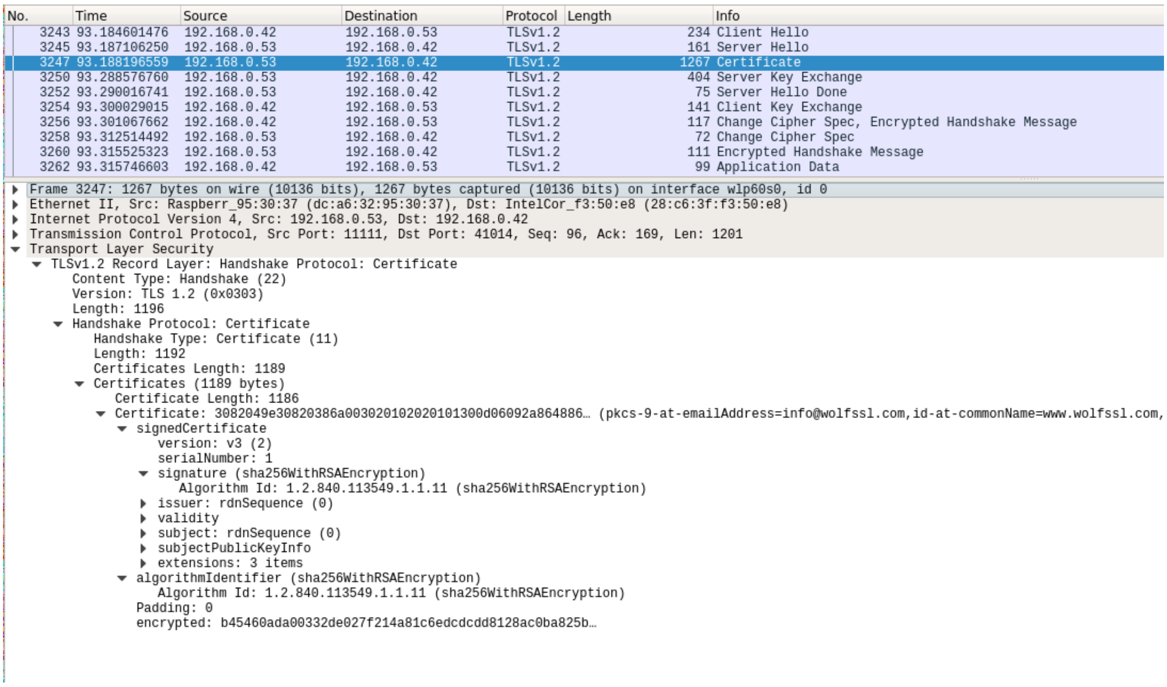


Figure 13: Server Certificate

The message **Server Key Exchange** is optional and sent when the public key present in the server's certificate is not suitable for key exchange, or if the cipher suite places a restriction requiring a temporary key. This key is used by the client to encrypt Client Key Exchange later in the process.

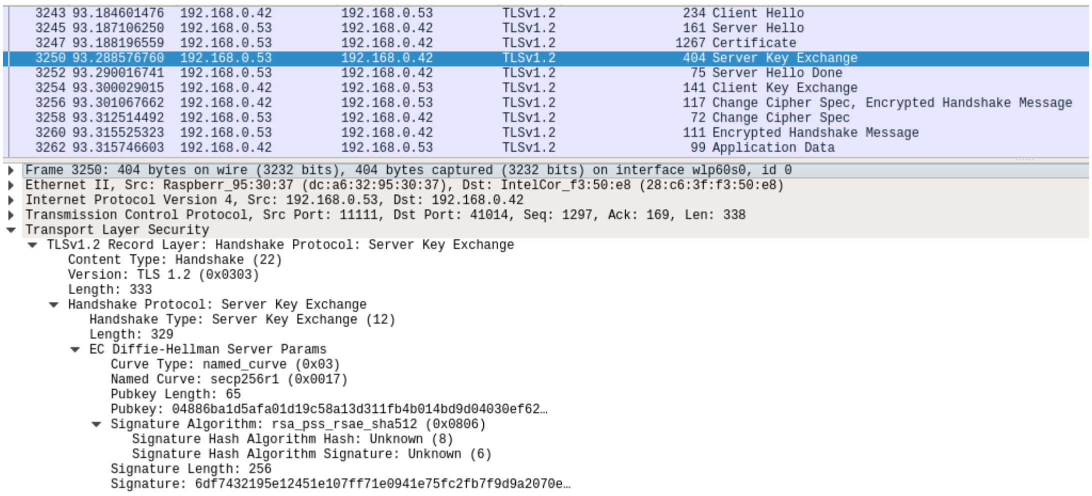


Figure 14: Server Key Exchange

This message indicates the server is done and is awaiting the client's response.

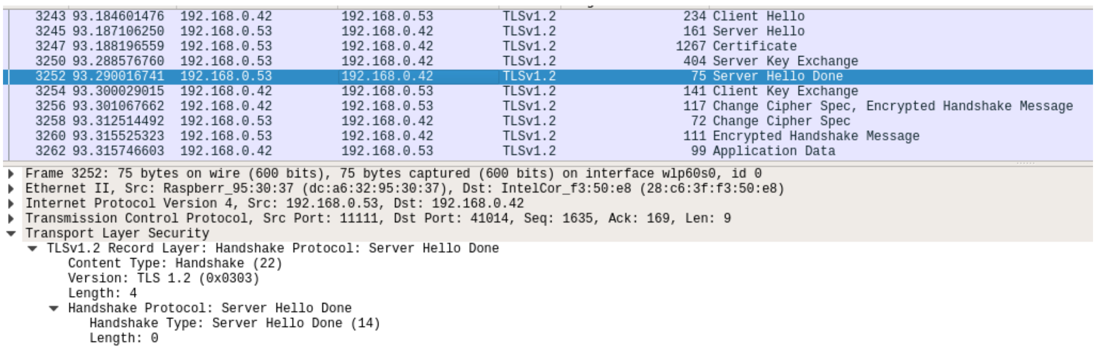


Figure 15: Server Hello Done

The **Client Key Exchange** message contains the protocol version of the client which the server verifies if it matches with the original client hello message. It also has the pre-master secret; it is a random number generated by the client and encrypted with the server public key. This along with the client and server random number is used to create the master secret. If the server can decrypt the message using the private key and can create the master secret locally, then the client is assured that the server has authenticated itself.

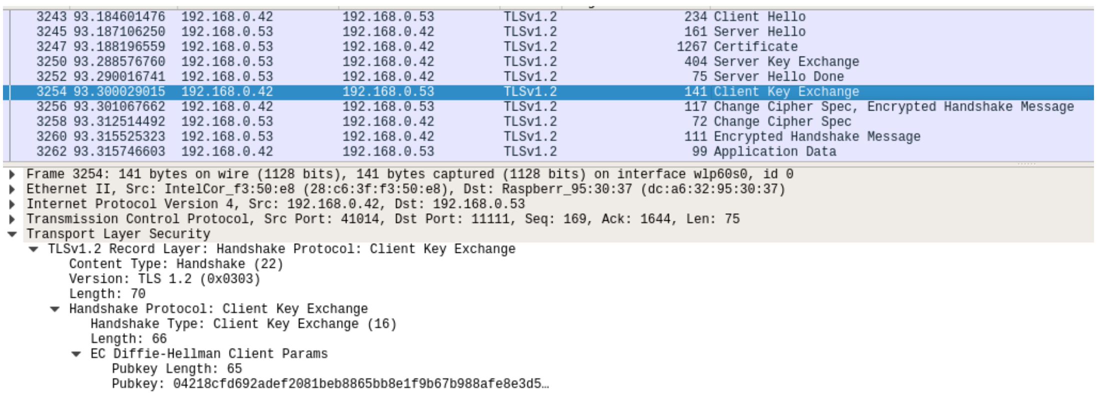


Figure 16: Client Key Exchange

The **Change Cipher Spec** message notifies the server that all the future messages will be encrypted using the algorithm and keys that were just negotiated.

The **Encrypted Handshake** message indicates that the TLS negotiations is completed for the client.

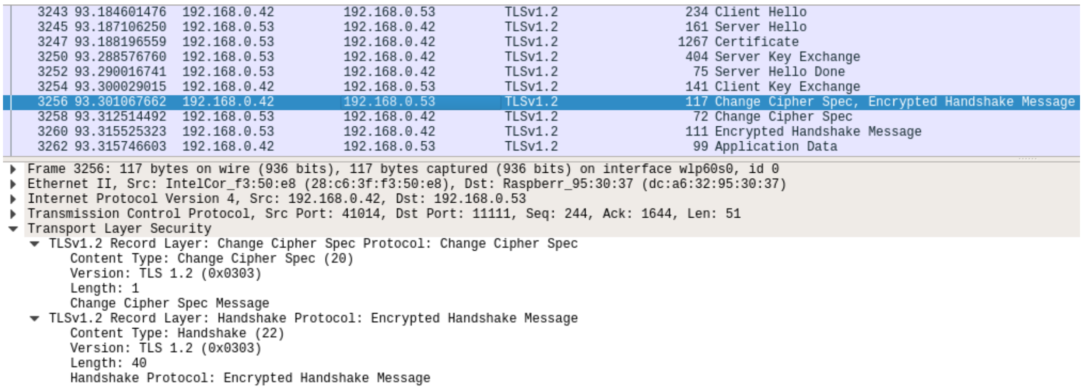


Figure 17: Change Cipher Spec and Encrypted Handshake

(Change Cipher Spec) The server informs the client that the messages will be encrypted with the existing algorithms and keys.

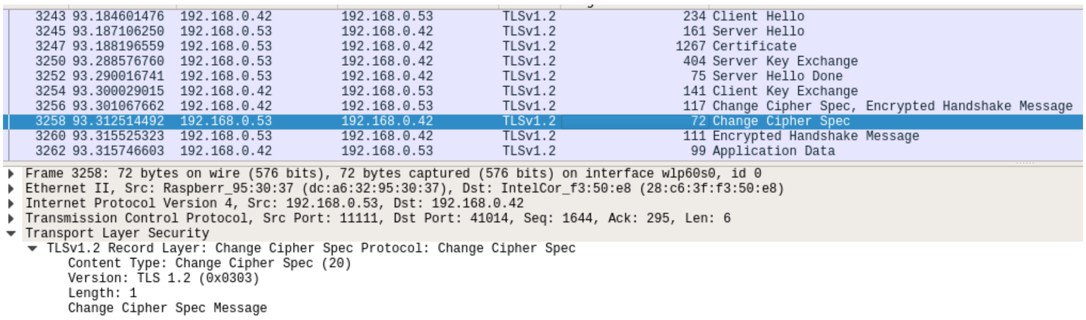


Figure 18: Change Cipher Spec

(Encrypted Handshake Message) Server informs the client the end of the TLS negotiations. It is like the client finished message.

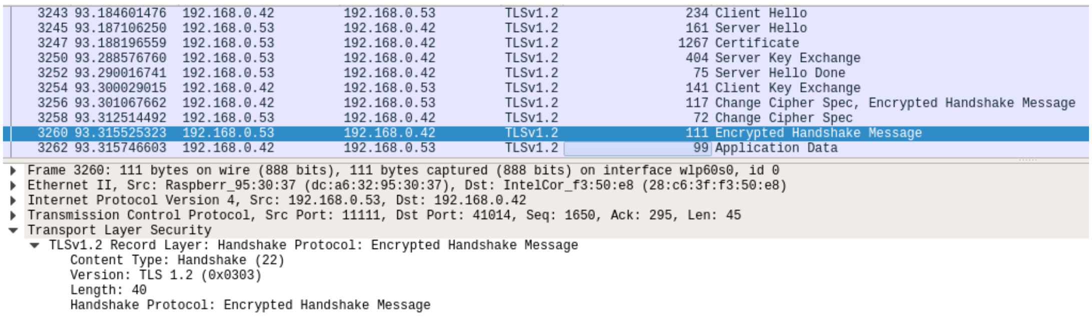


Figure 19: Encrypted Handshake

Every client will do the same handshake described above. After all clients' SSL handshake the situation in the server GUI is:

```
|--------------------------WolfSSL chat------------------------|
[System] Client Luca connected successfully
[System] ***************************
[System] Client Maria connected successfully
[System] ***************************
[System] Client Carlos connected successfully
[System] ***************************
--------------------------------------------------------------|
```

Figure 20: Connected clients, server GUI

To test the exact function of the programs, I sent several messages.

```
|--------------------------WolfSSL chat------------------------|
[Luca] Hi
[Maria] Hi Luca!
[Carlos] Hi everyone!
[Luca] This message is public
[Luca] list
[Server] Connected clients:
[1] Maria
[2] Carlos
[Luca] #1 this message is only for you, Maria!
[private-Maria] Thank you Luca
--------------------------------------------------------------|
```

Figure 21: Client Luca, GUI

```
|--------------------------WolfSSL chat------------------------|
[Luca] Hi
[Maria] Hi Luca!
[Carlos] Hi everyone!
[Luca] This message is public
[private-Luca] this message is only for you, Maria!
[Maria] list
[Server] Connected clients:
[0] Luca
[2] Carlos
[Maria] #0 Thank you Luca
[private-Maria] Thank you Luca
--------------------------------------------------------------|
```

Figure 22: Client Maria, GUI

```
|--------------------------WolfSSL chat------------------------|
[Luca] Hi
[Maria] Hi Luca!
[Carlos] Hi everyone!
[Luca] This message is public
---------------------------------------------------------------|
```

Figure 23: Client Carlos, GUI

With Wireshark you can see the difference between the TCP and TLS packets:

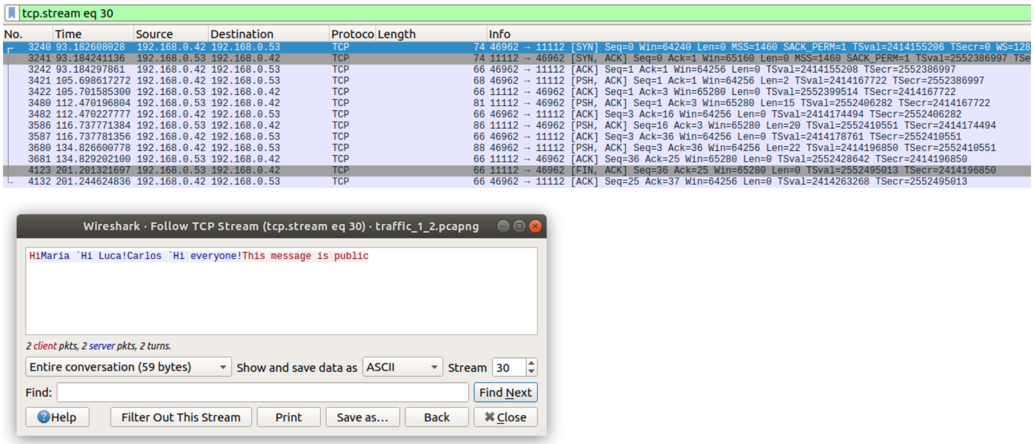


Figure 24: TCP traffic

In the TLS capture, after the handshake for each client, the messages are encrypted.

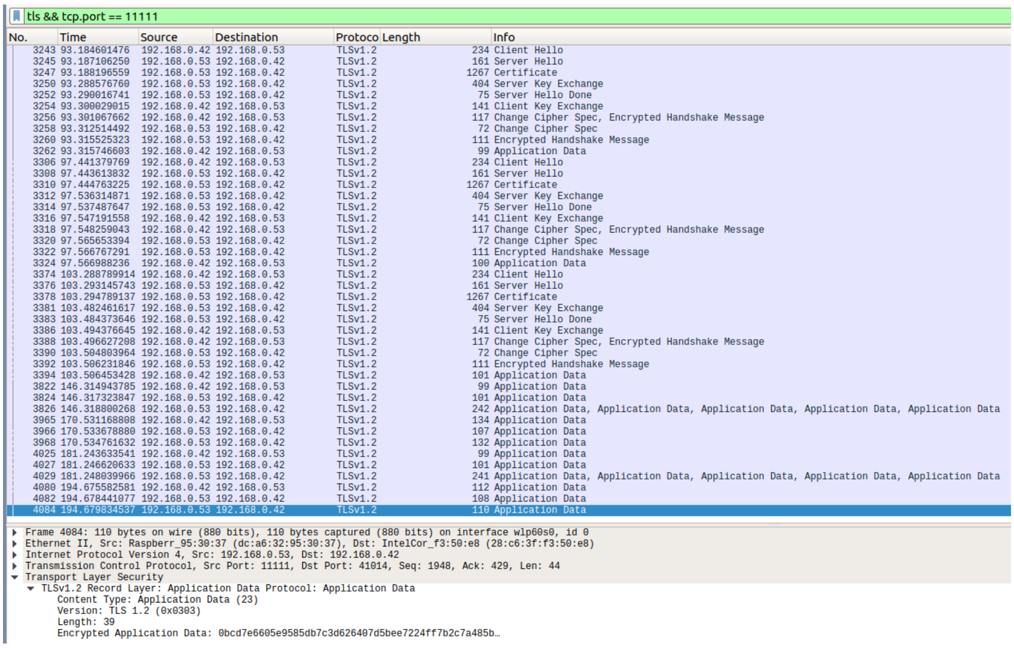


Figure 25: TLS traffic

5 Differences between WolfSSL and OpenSSL

5.1 Differences

The main differences are:

- Memory Usage
  - WolfSSL can be up to 20 times smaller than OpenSSL; The build size is between 20 and 100 KB and the runtime memory usage between 1 and 36 KB. This gives a major advantage of integrating in smaller embedded devices.
- Hardware Crypto
  - WolfSSL has a partnership with the most MCU manufacturers which allows to be quite early in the market to support hardware acceleration on huge list of platforms.
- Portability
  - WolfSSL is more portable than OpenSSL because is made for real-time, mobile, embedded and enterprise systems.

5.2 Build size and speed test

I installed openSSL (version: 3.0.0-alpha7-dev) and wolfSSL (version: 4.5.0) in a Raspberry Pi 4 and I could see the difference in occupancy on disk: OpenSSL occupies 412.5 MB while wolfSSL only 47 MB. This difference also affects the compilation and installation time; WolfSSL is compiled and installed in about 3 minutes while openSSL takes at least 10 minutes.

Wolfssl is definitely made for embedded systems, but how much can the data forwarding performance differ?
To answer this question I created two program that exchange data between a Raspberry and a laptop, one for wolfSSL and one for openSSL. In the specification, one of the two reads a file and sends the read content to the other one. The two programs are identical, the only difference are the function calls. One program has the functions of openssl and the other of wolfssl.
For the wolfSSL I used the same certificates and key used in the previous program, while for the openSSL I created a root CA certificate, a server key and a certificate signing request with these commands:

```
openssl genrsa -des3 -out CA-key.pem 2048
openssl req -new -key CA-key.pem -x509 -days 1000 -out CA-cert.pem
openssl genrsa -des3 -out server-key.pem 2048
openssl req ?new ?config openssl.cnf ?key server-key.pem ?out
    signingReq.csr
openssl x509 -req -days 365 -in signingReq.csr -CA CA-cert.pem
    -CAkey CA-key.pem -CAcreateserial -out server-cert.pem
```

Listing 12: openSSL commands

I created these programs to measure the data forwarding time.

The functions under examination are writefile (receive data from the secure channel) and sendfile (send data through the secure channel):

```
void sendfile(FILE *fp, SSL *ssl)
{
    int n;
    char sendline[MAX_LINE];
    clock_t t, sum = 0;
    t = clock();
    while ((n = fread(sendline, sizeof(char), MAX_LINE - 1, fp)) >
        0)
    {
        if (n != MAX_LINE && ferror(fp))
        {
            perror("Read File Error");
```

```
            exit(1);
        }
        t = clock();
        if ((total += SSL_write(ssl, sendline, strlen(sendline))) <
            0)
        {
            perror("Can't send file");
            exit(1);
        }
        sum += clock() - t;
        memset(sendline, 0, MAX_LINE);
    }
    double time_taken = ((double)sum) / CLOCKS_PER_SEC; // in
        seconds
    printf("%f seconds to send data \n", time_taken);
}
```

Listing 13: sendfile openSSL function

```
void writefile(SSL *ssl, FILE *fp)
{
    ssize_t n;
    char buff[MAX_LINE] = {0};
    clock_t t,sum = 0;
    t = clock();
    while ((n = SSL_read(ssl, buff, sizeof(buff))) > 0)
    {
        sum += clock() - t;
        total += n;
        if (n == -1)
        {
            perror("Receive File Error");
            exit(1);
        }
        if (fwrite(buff, sizeof(char), n, fp) != n)
        {
            perror("Write File Error");
            exit(1);
        }
        memset(buff, 0, MAX_LINE);
        t = clock();
    }
    double time_taken = ((double)sum) / CLOCKS_PER_SEC; // in
```

```
        seconds
    printf("%f seconds to receive data \n", time_taken);
}
```

Listing 14: writefile openSSL function

```
void sendfile(FILE *fp, WOLFSSL *ssl)
{
    int n;
    char sendline[MAX_LINE];
    clock_t t, sum = 0;
    t = clock();
    while ((n = fread(sendline, sizeof(char), MAX_LINE - 1, fp)) >
        0)
    {
        if (n != MAX_LINE && ferror(fp))
        {
            perror("Read File Error");
            exit(1);
        }
        t = clock();
        if ((total += wolfSSL_write(ssl, sendline,
            strlen(sendline))) < 0)
        {
            perror("Can't send file");
            exit(1);
        }
        sum += clock() - t;
        memset(sendline, 0, MAX_LINE);
    }
    double time_taken = ((double)sum) / CLOCKS_PER_SEC; // in
        seconds
    printf("%f seconds to send data \n", time_taken);
}
```

Listing 15: sendfile wolfSSL function

```
void writefile(WOLFSSL *ssl, FILE *fp)
{
    ssize_t n;
    char buff[MAX_LINE] = {0};
    clock_t t,sum = 0;
    t = clock();
```

```
    while ((n = wolfSSL_read(ssl, buff, sizeof(buff))) > 0)
    {
        sum += clock() - t;
        total += n;
        if (n == -1)
        {
            perror("Receive File Error");
            exit(1);
        }
        if (fwrite(buff, sizeof(char), n, fp) != n)
        {
            perror("Write File Error");
            exit(1);
        }
        memset(buff, 0, MAX_LINE);
        t = clock();
    }
    double time_taken = ((double)sum) / CLOCKS_PER_SEC; // in
        seconds
    printf("%f seconds to receive data \n", time_taken);
}
```

Listing 16: writefile wolfSSL function

The rest of the code is on github.

For the following test I created 4 files size about 128 MB, 256 MB, 512 MB and 1 GB with these commands:

```
128 MB = ./random.sh 128000000
256 MB = ./random.sh 256000000
512 MB = ./random.sh 512000000
1 GB = ./random.sh 1000000000
```

Listing 17: openSSL commands

This test is made in my gigabit home network. The time is in seconds. The cipher suite used is: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

Send file:

openSSL	time1	time2	time3	average
128 MB	0.18	0.25	0.21	0.21
256 MB	0.56	0.42	0.45	0.48
512 MB	1.13	1.06	0.77	0.99
1 GB	1.74	1.69	1.53	1.65

wolfSSL	time1	time2	time3	average
128 MB	5.40	5.36	4.64	5.13
256 MB	10.59	8.69	8.89	9.39
512 MB	20.73	20.88	22.16	21.26
1 GB	40.94	36.84	35.58	37.79

Receive file:

openSSL	time1	time2	time3	average
128 MB	3.61	3.89	3.59	3.70
256 MB	7.08	7.30	7.24	7.21
512 MB	15.73	16.65	14.56	15.65
1 GB	29.86	29.26	28.42	29.18

wolfSSL	time1	time2	time3	average
128 MB	12.53	13.33	13.83	13.23
256 MB	25.09	25.02	25.05	25.05
512 MB	51.15	50.83	53.30	51.76
1 GB	97.85	97.78	98.62	98.08

OpenSSL is more performing than wolfssl; the difference of time is relevant. Sending large amounts of data you can see the difference between them, but wolfSSL being born for embedded systems works very well and above all it takes up less space.

5.3 Run-time memory occupation

To compare the run-time memory occupation between openSSL and wolfSSL I used the linux task manager and valgrind. Using the task manager, the values aren't the exactly run-time memory occupation in a precise moment but the memory reserved by the operating system for the process; this allows me to do an estimation and a comparison among them.

In this comparison I monitored the RSS and the VSZ: the RSS is the Resident Set Size and is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory. It does include all stack and heap memory. The VSZ is the Virtual Memory Size. It includes all memory that the process can access, including memory that is swapped out, memory that is allocated, but not used, and memory that is from shared libraries.

The run-time memory occupation of the receive program (writefile function) is:

	RSS	VSZ
wolfSSL	1.3 MB	12.6MB
openSSL	5.9 MB	18.2MB

With valgrind openSSL has the total heap usage: 23,947 allocs, 23,947 frees, 6,937,196 bytes allocated and wolfSSL the total heap usage: 104 allocs, 104 frees, 121,036 bytes allocated.

Even in a small program the difference is relevant.

5.4 Segment size comparison

In order to compare the segment size between wolfSSL and openSSL, I used the previous send file program. To see the segment size I sent several files with openSSL and wolfSSL and I monitored the traffic data with Wireshark. Sending several files, I noticed that the segment size is always the same, either if I used openSSL or wolfSSL.

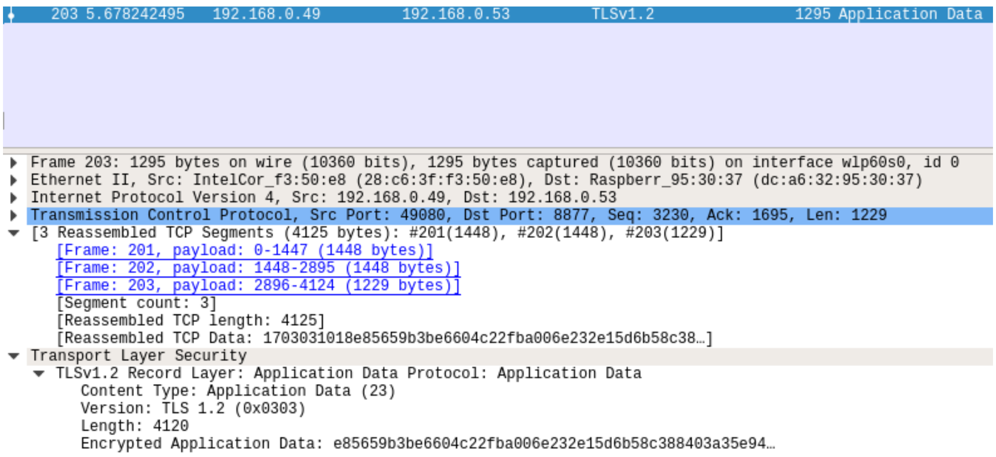


Figure 26: Application data wolfSSL

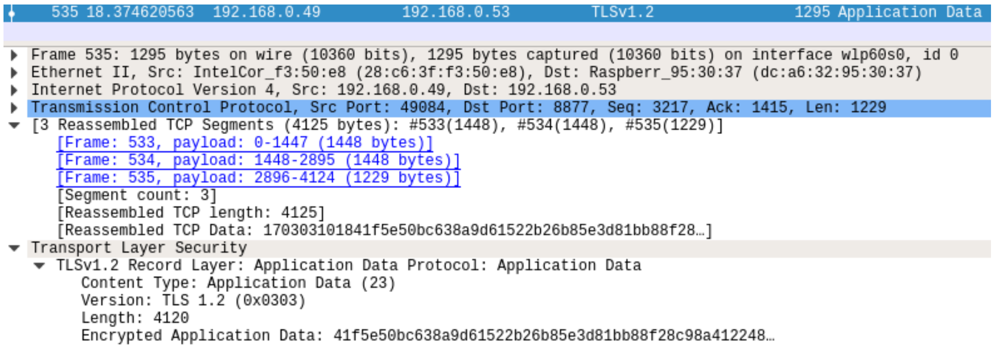


Figure 27: Application data openSSL

6 Conclusion

WolfSSL is a beautiful very well done library. They have a forum where thousand of users report their problems and solutions, in which employees too are very active.

A problem of wolfSSL manual is the lack of some details in some sections, such as in the difference with openSSL and its self-promotion orientation.

I think that this thesis can be a good starter guide to get information about wolfSSL since it is difficult to find material from the non official channels.

It is actively being used in a wide range of markets and products including the smart grid, IoT, industrial automation, connected home, M2M, auto industry, games, applications, databases, sensors, VoIP, routers, appliances, cloud services, and more; since wolfSSL is widely employed at a corporate level, there is a great lack of open source projects on the net.

In the future I intend to carry on this research, also focusing on wolfSSL's other products, such as wolfSSH, wolfCrypt, etc.

References

[1] Official WolfSSL repository
`https://github.com/wolfSSL/wolfssl`

[2] Official WolfSSL manual

[3] William Stallings, "Cryptography and Network Security Principles and Practice, Global Edition-Pearson (2017)", ISBN: 978-1292158587

[4] Rolf Oppliger, "SSL and TLS: Theory and Practice", ISBN: 978-1596934474

[5] Wiley, "Communicating Embedded Systems: Networks Applications", ISBN: 978-1848211445